```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// Borland.Vcl.Design.Proxies.txt
// Copyright (c) 2004. Borland Software Corporation All Rights Reserved.
{ ******************************************************* }
{                                                         }
{ Delphi/NET Runtime Library                              }
{                                                         }
{          }
{                                                         }
{ All Rights Reserved.                                    }
{                                                         }
{ ******************************************************* }


unit Borland.Vcl.Design.Proxies;

interface

uses
  System.Collections, System.Reflection, System.Reflection.Emit,
  System.Globalization, TypInfo, Classes, SysUtils;


//!! APIs have changed quite a bit
function CreateSubClass(AAncestor: TClass; const AClassName: string;
  const AUnitName: string = ''): TClass;
procedure DestroySubClass(AInstance: TObject); overload; deprecated;
procedure DestroySubClass(AClass: TClass); overload;
procedure RenameSubClass(AInstance: TObject; const AClassName: string;
  const AUnitName: string = ''); overload; deprecated;
procedure RenameSubClass(AClass: TClass; const AClassName: string;
  const AUnitName: string = ''); overload;

// TODO: ConstructSubClass - this should not be needed!
function ConstructSubClass(AClass: TClass; AParams: array of
TObject): TObject;
// TODO: ConstructComponent - this should not be needed!
function ConstructComponent(AClass: TComponentClass; AOwner:
TComponent = nil): TComponent;

function IsProxyClass(AInstance: TObject): Boolean; overload;
function IsProxyClass(AClass: TClass): Boolean; overload;

// TODO: ChangeToProxyClass, this can't work like the old way so will
this do?
procedure ChangeToProxyClass(AInstance: TObject{; TClass argument});
overload; deprecated;
procedure ChangeToProxyClass(AClass: TClass); overload;

function CreateSubClassMethod(AInstance: TObject;
  const AMethodName: string): TMethodCode;
procedure RenameSubClassMethod(AInstance: TObject;
  const AMethodCode: TMethodCode; const AMethodName: string);
procedure DestroySubClassMethod(AInstance: TObject;
  const AMethodCode: TMethodCode);

procedure HandleNotification(Sender: TObject; AComponent: TComponent;
Operation: TOperation);

procedure SaveIt;

type
```

```
  EProxyError = class(Exception);

implementation

uses System.Runtime.InteropServices;

type
  TProxyIntercept = class(TObject, IProxySystemSupport, IProxyTypInfoSupport)
  strict private
    function GetMethodAddress(AClass: TClass; const AName: string; out ACode: TMethodCode): Boolean;

    function GetMethodProp(AInstance: TObject; APropInfo: TPropInfo; out AMethod: TMethod): Boolean;
    function SetMethodProp(AInstance: TObject; APropInfo: TPropInfo; const AMethod: TMethod): Boolean;
    function GetUnitName(ATypeInfo: TTypeInfo; out AUnitName: string): Boolean;
  end;

  TInstanceRef = class(TObject)
  public
    Props: Hashtable;
    constructor Create;
  end;

  TProxyType = class(TypeDelegator)
  strict private
    class var
      FAssemblyBuilder: AssemblyBuilder;
      FModuleBuilder: ModuleBuilder;
      FProxyTypeIndex: Integer;
      FProxyIntercept: TProxyIntercept;
      FRootMetaType: System.Type;
      FRootHandleField: FieldInfo;
      FRootParentField: FieldInfo;
      FProxyNotificationMethod: MethodInfo;
      FSendNotificationMethod: MethodInfo;
      FProxies: Hashtable;
      FInstances: Hashtable;

    var
      FClassName: string;
      FUnitName: string;
      FMethods: Hashtable;

  strict protected
    class procedure CreateBoolAttribute(ATypeBuilder: TypeBuilder;
      AAttribute: System.Type; AValue: Boolean = True);
    class function CreateMetaSubType(ABaseType, AType: System.Type;
      ATypeBuilder: TypeBuilder): System.Type;
    class procedure CodeGenConstructors(ABaseType: System.Type; ATypeBuilder: TypeBuilder);
    class procedure CodeGenNotification(ABaseType: System.Type; ATypeBuilder: TypeBuilder);
    class function FindRealType(var AType: System.Type): Boolean;
  public
    class constructor Create;
    constructor Create(Ancestor: System.Type; const AClassName, AUnitName: string);

    // delegator work
    function get_FullName: string; override;
    function get_Name: string; override;
    function get_Namespace: string; override;

    // support for the public functions
    class function FindProxy(AInstance: TObject): TProxyType;
    function CreateMethod(const AMethodName: string): TMethodCode;
    procedure RenameMethod(const AMethodCode: TMethodCode; const AMethodName: string);
    procedure DestroyMethod(const AMethodCode: TMethodCode);

    // type versions of the public functions
    class function IsSubTyped(AType: System.Type): Boolean;
    class function CreateSubType(ABaseType: System.Type; const AClassName: string;
      const AUnitName: string = ''): System.Type;
    class procedure ChangeToProxyType(AType: System.Type);
    class procedure DestroySubType(AType: System.Type);
    class procedure RenameSubType(AType: System.Type; const AClassName: string;
      const AUnitName: string = '');

    // support functions for TProxyIntercept
    class function GetMethodAddress(AClass: TClass; const AName: string; out ACode: TMethodCode): Boolean;
    class function GetMethodProp(AInstance: TObject; APropInfo: TPropInfo; out AMethod: TMethod): Boolean;
    class function SetMethodProp(AInstance: TObject; APropInfo: TPropInfo; const AMethod: TMethod): Boolean;
    class function GetUnitName(ATypeInfo: TTypeInfo; out AUnitName: string): Boolean;

    class procedure HandleNotification(Sender: TObject; AComponent: TComponent; Operation: TOperation); static;

    // onetime snapshot of Proxies' scratch assembly
    // WARNING: once you 'SaveIt'; you can't create anymore proxy classes/types
    class procedure SaveIt;
  end;

  TObjects = array of TObject;
  TMethodProxy = class(TMethodCode)
  strict private
    FProxyType: TProxyType;
    FName: string;
  public
    constructor Create(AProxyType: TProxyType; const AName: string);
    procedure Clear;

    // TMethodProxy stuff
    procedure Rename(Value: string);
    function get_ProxyType: TProxyType;
    property ProxyType: TProxyType read get_ProxyType;

    // MemberInfo stuff
    function GetCustomAttributes(AInherit: Boolean): TObjects; override;
    function GetCustomAttributes(AttributeType: System.Type; Inherit: Boolean): TObjects; override;
    function IsDefined(AttributeType: System.Type; Inherit: Boolean): Boolean; override;
    function get_DeclaringType: System.Type; override;
    function get_MemberType: MemberTypes; override;
    function get_Name: string; override;
    function get_ReflectedType: System.Type; override;
    property DeclaringType: System.Type read get_DeclaringType;
    property MemberType: MemberTypes read get_MemberType;
    property Name: string read get_Name;
    property ReflectedType: System.Type read get_ReflectedType;
  end;

{ TProxyIntercept }

function TProxyIntercept.GetMethodAddress(AClass: TClass; const AName: string; out ACode: TMethodCode): Boolean;
begin
  Result := TProxyType.GetMethodAddress(AClass, AName, ACode);
end;

function TProxyIntercept.GetMethodProp(AInstance: TObject; APropInfo: TPropInfo; out AMethod: TMethod): Boolean;
begin
  Result := TProxyType.GetMethodProp(AInstance, APropInfo, AMethod);
end;

function TProxyIntercept.SetMethodProp(AInstance: TObject; APropInfo: TPropInfo; const AMethod: TMethod): Boolean;
begin
  Result := TProxyType.SetMethodProp(AInstance, APropInfo, AMethod);
end;

function TProxyIntercept.GetUnitName(ATypeInfo: TTypeInfo; out AUnitName: string): Boolean;
begin
  Result := TProxyType.GetUnitName(ATypeInfo, AUnitName);
end;

{ TInstanceRef }

constructor TInstanceRef.Create;
begin
  inherited;
  Props := Hashtable.Create;
end;


{ TProxyType }

const
  STestAssemblyName = 'VclDesignTime_ProxyAssembly';
  STestModuleName = 'VclDesignTime_ProxyModule';
  STestTypeName = 'Borland.Vcl.DesignTime.ProxyType%d';
  STestFileName = STestAssemblyName + '.dll';

var
  EchoLevel: Integer = 0;

{procedure EchoType(const APrefix: string; AType: System.Type; AMaxDepth: Integer = 4);
begin
  Inc(EchoLevel);
  try
    WriteLn(APrefix, ' ****************');
    if EchoLevel > AMaxDepth then
      WriteLn(APrefix, ' IS TOO DEEP')
    else
      if AType = nil then
```

```
        WriteLn(APrefix, ' IS NIL')
      else
      begin
        WriteLn(APrefix, '.Name = ', AType.Name);
        WriteLn(APrefix, '.FullName = ', AType.FullName);
        WriteLn(APrefix, '.Assembly = ', AType.Assembly.FullName);
        WriteLn(APrefix, '.AssemblyQualifedName = ', AType.AssemblyQualifiedName);
        WriteLn(APrefix, '.NameSpace = ', AType.NameSpace);
        WriteLn(APrefix, '.Attributes = ', System.Enum(AType.Attributes).ToString);
        WriteLn(APrefix, '.MemberType = ', System.Enum(AType.MemberType).ToString);
        try
          WriteLn(APrefix, '.TypeHandle = ', AType.TypeHandle.Value);
        except
          on E: Exception do
            WriteLn(APrefix, '.TypeHandle = ', E.Message);
        end;
        try
          WriteLn(APrefix, '.ClassName = ', AType.ClassName);
          WriteLn(APrefix, '.ClassInfo.Name = ', AType.ClassInfo.Name);
        except
          on E: Exception do
          begin
            WriteLn(APrefix, '.ClassName = ', E.Message);
            WriteLn(APrefix, '.ClassInfo.Name = ', E.Message);
          end;
        end;
        if AType.Module <> nil then
          WriteLn(APrefix, '.Module.Name = ', AType.Module.Name);
        if AType.BaseType <> nil then
          EchoType(APrefix + '.BaseType', AType.BaseType, AMaxDepth);
        if AType.DeclaringType <> nil then
          EchoType(APrefix + '.DeclaringType', AType.DeclaringType, AMaxDepth);
        if AType.UnderlyingSystemType <> nil then
          EchoType(APrefix + '.UnderlyingSystemType', AType.UnderlyingSystemType, AMaxDepth);
      end;
  finally
    Dec(EchoLevel);
    if EchoLevel < 0 then
    begin
      WriteLn('###################### How did that happen? ######################');
      EchoLevel := 0;
    end;
  end;
end;}

resourcestring
  SNoHandleNotification = 'Could not find Borland.Vcl.Design.Proxies.Unit.HandleNotification';
  SNoSendNotification = 'Could not find Borland.Vcl.Classes.Unit.SendNotification';
  SCouldNotFindBaseMeta = 'Could not find BaseMetaClass';
  SCouldNotFindTypeHandle = 'Could not find RootMetaClass.FInstanceTypeHandle';
  SCouldNotFindParent = 'Could not find RootMetaClass.FClassParent';
  SCouldNotFindConstructor = 'Could not find BaseType.Constructor';
  SCouldNotFindMetaConstructor = 'Could not find
```

```
MetaClass.Constructor';
  SAlreadyProxy = 'Type is already a proxy';
  STypeNotSubType = 'Type is not a subtype';
  SMethodNotMethodProxy = 'Method is not a method proxy';


class constructor TProxyType.Create;
var
  LAssemblyName: AssemblyName;
  LProxiesUnitType: System.Type;
  LClassesUnitType: System.Type;
begin
  // a place to work
  FProxies := Hashtable.Create;
  FInstances := Hashtable.Create;

  // create our scratcharea assembly and module
  LAssemblyName := AssemblyName.Create;
  LAssemblyName.Name := STestAssemblyName;
  FAssemblyBuilder := AppDomain.CurrentDomain.DefineDynamicAssembly(LAssemblyName, AssemblyBuilderAccess.RunAndSave);
  FModuleBuilder := FAssemblyBuilder.DefineDynamicModule(STestModuleName, STestFileName, True);

  // the following is need simply to keep the compiler from smartlinking certain functions into oblivion
  if FProxyTypeIndex < 0 then
  begin
    Borland.Vcl.Design.Proxies.HandleNotification(nil, nil, opInsert);
    Classes.SendNotification(nil, nil, opInsert);
  end;

  // find the sendnotification function over in Classes
  LProxiesUnitType := TypeOf(EProxyError).Assembly.GetType('Borland.Vcl.Design.Proxies.Unit');
  FProxyNotificationMethod := LProxiesUnitType.GetMethod('HandleNotification',
    BindingFlags.Public or BindingFlags.Static or BindingFlags.InvokeMethod);
  if FProxyNotificationMethod = nil then
    raise EProxyError.Create(SNoHandleNotification);

  // find the sendnotification function over in Classes
  LClassesUnitType := TypeOf(Classes.TOperation).Assembly.GetType('Borland.Vcl.Classes.Unit');
  FSendNotificationMethod := LClassesUnitType.GetMethod('SendNotification',
    BindingFlags.Public or BindingFlags.Static or BindingFlags.InvokeMethod);
  if FSendNotificationMethod = nil then
    raise EProxyError.Create(SNoSendNotification);

  // wedge into System and TypInfo
  FProxyIntercept := TProxyIntercept.Create;
  Borland.Delphi.System.ProxySystemSupport := FProxyIntercept;
  ProxyTypInfoSupport := FProxyIntercept;
end;

constructor TProxyType.Create(Ancestor: System.Type; const AClassName,
  AUnitName: string);
begin
  inherited Create(Ancestor);
  FClassName := AClassName;
  FUnitName := AUnitName;
  FMethods := Hashtable.Create;
end;

function TProxyType.get_Name: string;
begin
  Result := FClassName;
end;

function TProxyType.get_FullName: string;
begin
  Result := FUnitName + '.' + FClassName;
end;

function TProxyType.get_Namespace: string;
begin
  Result := FUnitName;
end;

class function TProxyType.IsSubTyped(AType: System.Type): Boolean;
begin
  // while FindRealType will change the AType we passed it the
  //   callee won't see it
  Result := FindRealType(AType);
end;

class procedure TProxyType.CreateBoolAttribute(ATypeBuilder: TypeBuilder;
  AAttribute: System.Type; AValue: Boolean);
var
  LAttributeConstructor: ConstructorInfo;
begin
  LAttributeConstructor := AAttribute.GetConstructor([TypeOf(AValue)]);
  ATypeBuilder.SetCustomAttribute(CustomAttributeBuilder.Create( LAttributeConstructor, [AValue]));
end;

class function TProxyType.CreateMetaSubType(ABaseType, AType: System.Type; ATypeBuilder: TypeBuilder): System.Type;
var
  LBaseType: System.Type;
  LTypeBuilder: TypeBuilder;
  LBaseConstructor: ConstructorInfo;
  LRootMetaType: System.Type;
  LRootHandleField: FieldInfo;
  LRootParentField: FieldInfo;
  LConstructorBuilder: ConstructorBuilder;
  LILGenerator: ILGenerator;
  LBaseInstanceField: FieldInfo;
  LInstanceField: FieldInfo;
  LTypeConstructorBuilder: ConstructorBuilder;

begin
  // find the base metatypes
  LBaseType := ABaseType.GetNestedType('@Meta' + ABaseType.Name);
  if LBaseType = nil then
    raise EProxyError.Create(SCouldNotFindBaseMeta);

  // found the root metatype yet?
  if FRootMetaType = nil then
  begin

    // chase up the metaclass parentage to find the root
    FRootMetaType := LBaseType;
    while FRootMetaType.BaseType <> nil do
    begin
      if FRootMetaType.BaseType = TypeOf(TObject) then
        break;
      FRootMetaType := FRootMetaType.BaseType;
    end;

    // look for a couple of fields that we will need
    FRootHandleField := FRootMetaType.GetField('FInstanceTypeHandle', BindingFlags.NonPublic or BindingFlags.Instance);
    if FRootHandleField = nil then
      raise EProxyError.Create(SCouldNotFindTypeHandle);

    FRootParentField := FRootMetaType.GetField('FClassParent', BindingFlags.NonPublic or BindingFlags.Instance);
    if FRootParentField = nil then
      raise EProxyError.Create(SCouldNotFindParent);
  end;

  // add a metatype for this type we are working on and add a field to the type
  LTypeBuilder := ATypeBuilder.DefineNestedType('@Meta' + ATypeBuilder.Name,
    TypeAttributes.NestedPublic or TypeAttributes.BeforeFieldInit, LBaseType);

  // add attribute or two
  CreateBoolAttribute(LTypeBuilder, TypeOf(System.CLSCompliantAttribute));
  CreateBoolAttribute(LTypeBuilder, TypeOf(System.Runtime.InteropServices.ComVisibleAttribute));

  // create our own instance field
  LInstanceField := LTypeBuilder.DefineField('@Instance', LTypeBuilder,
    FieldAttributes.Public or FieldAttributes.Static);

  // build constructor
  LConstructorBuilder := LTypeBuilder.DefineConstructor(MethodAttributes.Public or MethodAttributes.HideBySig,
    CallingConventions.Standard, []);
  LILGenerator := LConstructorBuilder.GetILGenerator;
  with LILGenerator, OpCodes do
  begin
    // CODE TO BE GENERATED
    // inherited Create;
    // FInstanceTypeHandle := Self.TypeHandle;
    // FClassParent := {ParentClass}.@Instance; // only codegen if parentclass has one

    LBaseConstructor := LBaseType.GetConstructor([]);  // find the base's create
    if LBaseConstructor = nil then
      raise EProxyError.Create(SCouldNotFindConstructor);
    Emit(Ldarg_0);                                      // push the instance
    Emit(Call, LBaseConstructor);       // emit a call to the parent constructor

    Emit(Ldarg_0);                                      // push
```

```
the instance
    Emit(Ldtoken, AType);                          // push the hand
le
of the type
    Emit(Stfld, FRootHandleField);       // store the handle in th
e
root's field

    // see if the base metatype has an instance field yet
    LBaseInstanceField := LBaseType.GetField('@Instance',
BindingFlags.Public or BindingFlags.Static);
    if LBaseInstanceField <> nil then
    begin
      Emit(Ldarg_0);                                       // pus
h
the instance
      Emit(Ldsfld, LBaseInstanceField);                  // get t
he
parent info
      Emit(Stfld, FRootParentField);                // store it i
nto
root field
    end;

    Emit(Ret);

   // fini
  end;

  // now create the class constructor
  LTypeConstructorBuilder := LTypeBuilder.DefineTypeInitializer;
  LILGenerator := LTypeConstructorBuilder.GetILGenerator;
  with LILGenerator, OpCodes do
  begin
    // CODE TO BE GENERATED
    // @Instance := @Meta{Class}.Create;

    Emit(Newobj, LConstructorBuilder);    // create an instance of

the metaclass
    Emit(Stsfld, LInstanceField);              // store it in our
instance field

    Emit(Ret);

   // fini
  end;

  // before we leave we had better actually create the type hadn't
 we
  Result := LTypeBuilder.CreateType;
end;

class procedure TProxyType.CodeGenConstructors(ABaseType:
System.Type; ATypeBuilder: TypeBuilder);
var
  LConstructors: array of ConstructorInfo;
  LParameters: array of ParameterInfo;
  LParamTypes: array of System.Type;
  LConstructorBaseType: System.Type;
  LConstructorBuilder: ConstructorBuilder;
  LILGenerator: ILGenerator;
  LConstructorNdx, LParameterNdx: Integer;
begin
  LConstructorBaseType := ABaseType;
  while LConstructorBaseType <> nil do
  begin

```

```
    // see if it has any constructors
    LConstructors := LConstructorBaseType.GetConstructors;
    if Length(LConstructors) <> 0 then
    begin
      for LConstructorNdx := Low(LConstructors) to High(LConstructors) do
      begin
        with LConstructors[LConstructorNdx] do
        begin
          // copy the param and in turn their types
          LParameters := GetParameters;
          SetLength(LParamTypes, Length(LParameters));
          for LParameterNdx := Low(LParameters) to High(LParameters) do
            LParamTypes[LParameterNdx] := LParameters[LParameterNdx].ParameterType;

          // construct a constructor builder
          LConstructorBuilder := ATypeBuilder.DefineConstructor(Attributes,
            CallingConvention, LParamTypes);
        end;

        // lets write some code
        LILGenerator := LConstructorBuilder.GetILGenerator;
        with LILGenerator, OpCodes do
        begin
          // CODE TO BE GENERATED
          // inherited Create({arg count depends on parentclass})

          Emit(Ldarg_0);                                          // push instance
          for LParameterNdx := 1 to Length(LParameters) do
            Emit(Ldarg_S, LParameterNdx);                         // push params
          Emit(Call, LConstructors[LConstructorNdx]);       // call the base ctr

          Emit(Ret);                                              // fini
        end;
      end;

      // done
      break;
    end;

    // move up a level
    LConstructorBaseType := LConstructorBaseType.BaseType;
  end;
end;

class procedure TProxyType.CodeGenNotification(ABaseType: System.Type; ATypeBuilder: TypeBuilder);
var
  LParamTypes: array of System.Type;
  LBaseNotificationMethod: MethodInfo;
  LMethodBuilder: MethodBuilder;
  LILGenerator: ILGenerator;
  LLabel: System.Reflection.Emit.Label;
begin
  // get the param list ready
  SetLength(LParamTypes, 2);
  LParamTypes[0] := TypeOf(Classes.TComponent);
  LParamTypes[1] := TypeOf(Classes.TOperation);

  // see if we can find a notification method to call
  LBaseNotificationMethod := ABaseType.GetMethod('Notification',
    BindingFlags.Public or BindingFlags.NonPublic or BindingFlags.Instance or
    BindingFlags.InvokeMethod, nil, LParamTypes, nil);
  if LBaseNotificationMethod <> nil then
  begin

    // create a builder
    with LBaseNotificationMethod do
      LMethodBuilder := ATypeBuilder.DefineMethod(Name,
        MethodAttributes.FamORAssem or MethodAttributes.Virtual,
        CallingConvention, ReturnType, LParamTypes);

    // let's write some code!
    LILGenerator := LMethodBuilder.GetILGenerator;
    with LILGenerator, OpCodes do
    begin
      // CODE TO BE GENERATED
      // Borland.Vcl.Design.Proxies.HandleNotification(Self, AComponent, AOperation);
      // if Borland.Vcl.Classes.SendNotification(Self, AComponent, AOperation) then
      //   inherited Notification(AComponent, AOperation);

      Emit(Ldarg_0);                              // push instance
      Emit(Ldarg_1);                              // push component reference
      Emit(Ldarg_2);                              // push what is happening to it
      Emit(Call, FProxyNotificationMethod);       // call the proxy's notify-wedge

      Emit(Ldarg_0);                              // push instance
      Emit(Ldarg_1);                              // push component reference
      Emit(Ldarg_2);                              // push what is happening to it
      Emit(Call, FSendNotificationMethod);        // call classes' sendnotification

      LLabel := DefineLabel;
      Emit(Brfalse_S, LLabel);                    // if result is false then...

      Emit(Ldarg_0);                              // push instance
      Emit(Ldarg_1);                              // push component reference
      Emit(Ldarg_2);                              // push what is happening to it
      Emit(Call, LBaseNotificationMethod);        // call the base's method

      MarkLabel(LLabel);                          // ...jump to here
```

```

        Emit(Ret);

  // fini
    end;
  end;
end;

class function TProxyType.FindRealType(var AType: System.Type): Boolean;
begin
  // just in case were given a proxy type lets find the real type
  if AType is TProxyType then
    AType := AType.UnderlyingSystemType;

  // see if we can find it in our list
  Result := FProxies.Contains(AType);
end;

class function TProxyType.CreateSubType(ABaseType: System.Type;
  const AClassName: string; const AUnitName: string = ''): System.Type;
var
  LTypeBuilder: TypeBuilder;
  LMetaType: System.Type;
  LMetaConstructor: ConstructorInfo;
  LProxyType: TProxyType;
  LNewType: System.Type;
begin
  // find the real type...  if we have been handed a proxytype, instead of
  //   a 'realtype', then FindRealType will modify ABaseType so that
it
  //   points to the proxy's UnderlyingSystemType.
  FindRealType(ABaseType);

  // create a type builder         ...remember each type must have a
unique name
  LTypeBuilder := FModuleBuilder.DefineType(Format(STestTypeName, [FProxyTypeIndex]), TypeAttributes.Public, ABaseType);
  Inc(FProxyTypeIndex);

  // find the first ancestor class that has constructors and copy them
  CodeGenConstructors(ABaseType, LTypeBuilder);

  // TODO: If the type is a TComponent desendent then we need to hook
notification
  CodeGenNotification(ABaseType, LTypeBuilder);

  // quick make the type before it slips away again :-)
  LNewType := LTypeBuilder.CreateType;
  LProxyType := TProxyType.Create(LNewType, AClassName, AUnitName);

  // make up a metaclass for the Delphi System unit
  LMetaType := CreateMetaSubType(ABaseType, LNewType, LTypeBuilder);
  LMetaConstructor := LMetaType.GetConstructor([]);
  if LMetaConstructor = nil then
    raise EProxyError.Create(SCouldNotFindMetaConstructor);

  // plug ourselves into the class delegator system so that our proxy
type will
```

```
  //  be found when someone does a ClassInfo on this type/metatype
  SetClassDelegator(LProxyType, LMetaConstructor.Invoke([]));

  // add it to the list of known 'live' proxies
  FProxies.Add(LNewType, LMetaType);

  // return the proxy type
  Result := LProxyType;
end;

class procedure TProxyType.SaveIt;
begin
  // caution: this is a one shot thing! once you call this you can't
  //   create anymore proxy classes.
  FAssemblyBuilder.Save(STestFileName);
end;

class procedure TProxyType.ChangeToProxyType(AType: System.Type);
begin
  // if it is already a proxy then complain...  if we have been handed a
  //   proxytype, instead of a 'realtype', then FindRealType will modify
  //   AType so that it points to the proxy's UnderlyingSystemType.
  if FindRealType(AType) then
    raise EProxyError.Create(SAlreadyProxy);

  // add the delegator
  SetClassDelegator(TProxyType.Create(AType, AType.Name, AType.NameSpace));

  // add it the proxy list
  FProxies.Add(AType, TypeOf(TClass(AType)));
end;

class procedure TProxyType.DestroySubType(AType: System.Type);
begin
  // is it really subtyped?  if so then complain loudly...  if we have been
  //   handed a proxytype, instead of a 'realtype', then FindRealType will
  //   modify AType so that it points to the proxy's UnderlyingSystemType.
  if not FindRealType(AType) then
    raise EProxyError.Create(STypeNotSubType);

  // remove it from the proxy list
  FProxies.Remove(AType);

  // remove the delegator
  RemoveClassDelegator(AType);
end;

class procedure TProxyType.RenameSubType(AType: System.Type;
  const AClassName: string; const AUnitName: string = '');
begin
  // is it really subtyped? (we call IsSubType because we don't want the realtype)
  if not IsSubTyped(AType) then
    raise EProxyError.Create(STypeNotSubType);

  // change the name
  TProxyType(AType).FClassName := AClassName;
  if AUnitName <> '' then
```

```
    TProxyType(AType).FUnitName := AUnitName;
end;

class function TProxyType.FindProxy(AInstance: TObject): TProxyType;
var
  LType: System.Type;
begin
  // find the type
  LType := AInstance.ClassInfo;

  // make sure it is what we need otherwise complain
  if not (LType is TProxyType) then
    raise EProxyError.Create(STypeNotSubType);
  Result := TProxyType(LType);
end;

function TProxyType.CreateMethod(const AMethodName: string): TMethodCode;
var
  LMethodCode: TMethodCode;
begin
  LMethodCode := TMethodProxy(FMethods[AMethodName]);
  if LMethodCode = nil then
  begin
    LMethodCode := TMethodProxy.Create(Self, AMethodName);
    FMethods.Add(AMethodName, LMethodCode);
  end;
  Result := LMethodCode;
end;

procedure TProxyType.RenameMethod(const AMethodCode: TMethodCode; const AMethodName: string);
begin
  // make sure it is a method proxy
  if not (AMethodCode is TMethodProxy) then
    raise EProxyError.Create(SMethodNotMethodProxy);

  // remove, rename and re-add
  FMethods.Remove(AMethodCode.Name);
  TMethodProxy(AMethodCode).Rename(AMethodName);
  FMethods.Add(AMethodName, AMethodCode);
end;

procedure TProxyType.DestroyMethod(const AMethodCode: TMethodCode);
begin
  // make sure it is a method proxy
  if not (AMethodCode is TMethodProxy) then
    raise EProxyError.Create(SMethodNotMethodProxy);

  // remove and clear
  FMethods.Remove(AMethodCode.Name);
  TMethodProxy(AMethodCode).Clear;
end;

class function TProxyType.GetMethodAddress(AClass: TClass; const AName: string; out ACode: TMethodCode): Boolean;
var
  LType: System.Type;
begin
  // assume failure
  ACode := nil;

  // find the class' type
  LType := AClass.ClassInfo;
  Result := LType is TProxyType;

  // keep looking but only if the type is a TProxyType
  while LType is TProxyType do
  begin

    // see if there is a method
    ACode := TMethodCode(TProxyType(LType).FMethods.Item[AName]);
    if ACode <> nil then
      break;

    // still nothing?  then look at the parent class
    AClass := AClass.ClassParent;
    LType := AClass.ClassInfo;
  end;
end;

class function TProxyType.GetMethodProp(AInstance: TObject;
APropInfo: TPropInfo; out AMethod: TMethod): Boolean;
var
  LInstanceRef: TInstanceRef;
  LMethodRef: TObject;
begin
  // find the instance
  LInstanceRef := TInstanceRef(FInstances.Item[AInstance]);
  Result := LInstanceRef <> nil;

  // do our thing?
  if Result then
  begin

    // find the property
    LMethodRef := LInstanceRef.Props.Item[APropInfo];

    // if nothing
    if LMethodRef = nil then
      AMethod := TMethod.Empty
    else
      AMethod := TMethod(LMethodRef);

    // I guess it worked
    Result := True;
  end;
end;

class function TProxyType.SetMethodProp(AInstance: TObject;
APropInfo: TPropInfo; const AMethod: TMethod): Boolean;
var
  LInstanceRef: TInstanceRef;
begin
  // something we care about?
  Result := (AMethod.Data = nil) or IsProxyClass(AMethod.Data);
  if Result then
  begin

    // find the instance
    LInstanceRef := TInstanceRef(FInstances.Item[AInstance]);
    if LInstanceRef = nil then
    begin
      LInstanceRef := TInstanceRef.Create;
      FInstances.Add(AInstance, LInstanceRef);
    end;

    // adding?
    if not AMethod.IsEmpty then
      LInstanceRef.Props[APropInfo] := AMethod.Clone

    // removing?
    else
    begin
```

```
      // poof!
      LInstanceRef.Props.Remove(APropInfo);

      // if there are no props defined then get rid of the instanc
e
itself
      if LInstanceRef.Props.Count = 0 then
        FInstances.Remove(AInstance);
    end;
  end;
end;

class function TProxyType.GetUnitName(ATypeInfo: TTypeInfo; out
AUnitName: string): Boolean;
begin
  // assume success
  Result := True;

  // go find the right type and get its proxy, if there is one
  AUnitName := TClass(ATypeInfo).ClassInfo.NameSpace;
end;

class procedure TProxyType.HandleNotification(Sender: TObject;
AComponent: TComponent; Operation: TOperation);
begin
  // remove it from our list
  if Operation = opRemove then
    TProxyType.FInstances.Remove(AComponent);
end;

{ TMethodProxy }

constructor TMethodProxy.Create(AProxyType: TProxyType; const AName
e:
string);
begin
  inherited Create;
  FProxyType := AProxyType;
  FName := AName;
end;

procedure TMethodProxy.Clear;
begin
  FProxyType := nil;
  FName := '';
end;

procedure TMethodProxy.Rename(Value: string);
begin
  FName := Value;
end;

function TMethodProxy.get_ProxyType: TProxyType;
begin
  Result := FProxyType;
end;

function TMethodProxy.GetCustomAttributes(AInherit: Boolean):
TObjects;
begin
  Result := GetCustomAttributes(nil, AInherit);
end;

function TMethodProxy.GetCustomAttributes(AttributeType: System.Ty
pe;
Inherit: Boolean): TObjects;
begin
  SetLength(Result, 0);
```

```
end;

function TMethodProxy.IsDefined(AttributeType: System.Type; Inherit: Boolean): Boolean;
begin
  Result := False;
end;

function TMethodProxy.get_DeclaringType: System.Type;
begin
  Result := FProxyType;
end;

function TMethodProxy.get_MemberType: MemberTypes;
begin
  Result := MemberTypes.Method;
end;

function TMethodProxy.get_Name: string;
begin
  Result := FName;
end;

function TMethodProxy.get_ReflectedType: System.Type;
begin
  Result := nil;
end;

{ Unit functions }

function CreateSubClass(AAncestor: TClass; const AClassName: string;
  const AUnitName: string): TClass;
begin
  Result := TClass(TProxyType.CreateSubType(AAncestor.ClassInfo, AClassName, AUnitName));
end;

resourcestring
  SNoValidConstructor = 'No valid constructor found for %s.';

function ConstructSubClass(AClass: TClass; AParams: array of TObject): TObject;
var
  LParameterNdx: Integer;
  LParamTypes: array of System.Type;
  LConstructor: ConstructorInfo;
begin
  SetLength(LParamTypes, Length(AParams));
  for LParameterNdx := Low(AParams) to High(AParams) do
    if AParams[LParameterNdx] = nil then
      LParamTypes[LParameterNdx] := TypeOf(TObject)
    else
      LParamTypes[LParameterNdx] := AParams[LParameterNdx].ClassInfo;
  LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
  if LConstructor = nil then
    raise EProxyError.CreateFmt(SNoValidConstructor, [AClass.ClassName]);
  Result := LConstructor.Invoke(AParams)
end;

function ConstructComponent(AClass: TComponentClass; AOwner: TComponent = nil): TComponent;
var
  LParamTypes: array of System.Type;
  LConstructor: ConstructorInfo;
```

```
begin
//Result := AClass.Create(AOwner); // Corbin note: we need this to work...soon.....
//Exit;
  SetLength(LParamTypes, 1);
  LParamTypes[0] := TypeInfo(TComponent);
  LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
  if LConstructor = nil then
  begin
    { Try a parameterless constructor }
    SetLength(LParamTypes, 0);
    LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
    if LConstructor <> nil then
    begin
      Result := TComponent(LConstructor.Invoke([]));
      if AOwner <> nil then
         AOwner.InsertComponent(Result);
    end
    else
      raise EProxyError.CreateFmt(SNoValidConstructor, [AClass.ClassName]);
  end
  else
    Result := TComponent(LConstructor.Invoke([AOwner]))
end;

procedure DestroySubClass(AInstance: TObject);
begin
  DestroySubClass(AInstance.ClassType);
end;

procedure DestroySubClass(AClass: TClass);
begin
  TProxyType.DestroySubType(AClass.ClassInfo);
end;

procedure RenameSubClass(AInstance: TObject; const AClassName, AUnitName: string);
begin
  RenameSubClass(AInstance.ClassType, AClassName, AUnitName);
end;

procedure RenameSubClass(AClass: TClass; const AClassName, AUnitName: string);
begin
  TProxyType.RenameSubType(AClass.ClassInfo, AClassName, AUnitName);
end;

function IsProxyClass(AInstance: TObject): Boolean;
begin
  Result := IsProxyClass(AInstance.ClassType);
end;

function IsProxyClass(AClass: TClass): Boolean;
begin
  Result := TProxyType.IsSubTyped(AClass.ClassInfo);
end;

procedure ChangeToProxyClass(AInstance: TObject);
begin
  ChangeToProxyClass(AInstance.ClassType);
end;

procedure ChangeToProxyClass(AClass: TClass);
begin
```

```
  TProxyType.ChangeToProxyType(AClass.ClassInfo);
end;

function CreateSubClassMethod(AInstance: TObject; const AMethodName: string): TMethodCode;
begin
  Result := TProxyType.FindProxy(AInstance).CreateMethod(AMethodName);
end;

procedure RenameSubClassMethod(AInstance: TObject; const AMethodCode: TMethodCode; const AMethodName: string);
begin
  TProxyType.FindProxy(AInstance).RenameMethod(AMethodCode, AMethodName);
end;

procedure DestroySubClassMethod(AInstance: TObject; const AMethodCode: TMethodCode);
begin
  TProxyType.FindProxy(AInstance).DestroyMethod(AMethodCode);
end;

procedure HandleNotification(Sender: TObject; AComponent: TComponent; Operation: TOperation);
begin
  TProxyType.HandleNotification(Sender, AComponent, Operation);
end;

procedure SaveIt;
begin
  TProxyType.SaveIt;
end;

end.
```